NUSC Technical Report 8807
30 December 1990

12

DTIC FILE COPY

AD-A232 032

Mapping an Adaptable Eigenstructure Technique Onto the Topologix Hypercube

M. H. Leonhardt
Submarine Sonar Department

Naval Underwater Systems Center
Newport, Rhode Island•New London, Connecticut

PREFACE

This report was prepared under the New Professional Bid and Proposal Program, NUSC project number 791P15. The title of the project was *Implementation of Minimum Variance Distortionless Response Beamformer.*

The technical reviewer for the report was Dr. J. Muñoz.

The author would like to thank A. C. Barthelemy, W. R. Bernecky, T. C. Choinski, and Dr. J. Muñoz, all of NUSC, for providing helpful suggestions during the preparation of the manuscript.

REVIEWED AND APPROVED: 30 DECEMBER 1990

F. J. Kingsbury
Head, Submarine Sonar Department

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	30 December 1990	Final Oct 89 - Oct 90

4. TITLE AND SUBTITLE
MAPPING AN ADAPTABLE EIGENSTRUCTURE TECHNIQUE ONTO THE TOPOLOGIX HYPERCUBE

5. FUNDING NUMBERS
791P15

6. AUTHOR(S)
M. H. Leonhardt

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Underwater Systems Center
New London Laboratory
New London, Connecticut 06320

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 8807

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Eigenstructure methods provide accuracy and stability to signal processing applications, such as the enhanced minimum variance distortionless response (EMVDR) algorithm. Eigenstructure methods, however, require the high throughput that is only available in parallel computer systems. One such system is the hypercube multiprocessor, which, along with high throughput, also provides distributed memory and efficient networking. This report describes the implementation of an eigenstructure method on the Topologix hypercube architecture that will produce the eigenstructure required in the EMVDR beamformer.

14. SUBJECT TERMS
Eigenstructure
EMVDR
Hypercube Multiprocessor
Parallel Computer System
Signal Processing
Topologix Hypercube

15. NUMBER OF PAGES
52

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

ABF	Adaptive Beamforming
BLAS	Basic Linear Algebra Subroutines
CSDM	Cross-Spectral Density Matrix
EMVDR	Enhanced Minimum Variance Distortionless Response
LAPACK	Linear Algebra Package
LINPACK	Linear Algebra Package (revised)
Mbps	Million Bits Per Second
MFLOPs	Million Floating Point Operations Per Second
MIPs	Million Instructions Per Second
MUSIC	Multiple Emitter Signal Location Wavenumber Spectrum Estimator
MVDR	Minimum Variance Distortionless Response
SNR	Signal-to-Noise Ratio
VLSI	Very Large Scale Integration

LIST OF SYMBOLS

A	An $n \times n$ matrix (always large letter)
$\underline{x}$	A $1 \times n$ or an $n \times 1$ vector
A^H	Hermitian transpose of matrix A
$\underline{x}^H$	Hermitian transpose of vector $\underline{x}$
A_k	Matrix A at step k
$\underline{x}_i$	Element i of vector $\underline{x}$
$\|s\|$	Magnitude of scalar s
$\|\|\underline{x}\|\|$	Euclidean norm of vector $\underline{x}$
A^{mk}	Square submatrix A comprised of elements of A_{ij} for which $\frac{mG}{\sqrt{p}} \le i < \frac{(m+1)G}{\sqrt{p}}$, $\frac{kG}{\sqrt{p}} \le j < \frac{(k+1)G}{\sqrt{p}}$, G is number of rows in matrix A, and p is number of processors.

MAPPING AN ADAPTABLE EIGENSTRUCTURE TECHNIQUE ONTO THE TOPOLOGIX HYPERCUBE

INTRODUCTION

Linear algebra techniques, such as matrix-matrix multiplication and eigenstructure decomposition, have been studied on sequential computer systems for many years. Some of the algorithms used in these applications, most of which have been widely published, include BLAS I, II, and III; the Linear Algebra Package (LAPACK); and the revised Linear Algebra Package (LINPACK). In recent years, it has been determined that parallel implementations of the best sequential algorithms for linear algebra provide the optimal criteria for measuring the performance of future algorithms and for evaluating new parallel architectures. Such parallel implementations also provide accuracy and stability to signal processing applications.

BACKGROUND

The capability of adaptive beamforming (ABF) to maximize signal-to-noise ratio (SNR) is well-known. One of the many techniques is the minimum variance distortionless response (MVDR) algorithm.[1] First documented in the early 1960s, the MVDR technique was not immediately accepted because of its computationally intensive nature. This problem was solved with the arrival of very large scale integration (VLSI) technology, which increased the available processing power.

Previous MVDR implementations[2] showed that the algorithm becomes unstable for large SNR values. The instability occurs when correlation exists between sensor outputs, making the covariance matrix ill-conditioned. An ill-conditioned matrix (one where there is a large difference between smallest and largest eigenvalues) is noninvertible. Eigenstructure techniques, however, allow some correlation to exist without affecting the stability of the algorithm.[3]

RESOLUTION

Eigenstructure-based analysis methods, which evolved concurrently with MVDR since the early 1970s, have led to current high resolution procedures. Owsley[4] developed a single approach called the Enhanced MVDR (EMVDR) beamformer that used advantages from both techniques. This approach provides a compromise between high resolution direction-finding algorithms (such as the multiple emitter signal location wavenumber spectrum estimator (MUSIC)) and ABF algorithms (such as MVDR) by using estimated eigenstructures rather than the Cholesky factorizations of MVDR.

PARALLEL IMPLEMENTATION RATIONALE

As mentioned earlier, signal processing applications requiring linear algebra were limited by the throughput provided with sequential computer systems. The use of VLSI technology in sequential computers was still not enough to provide the required speed. However, new highly parallel architectures can support the high throughput with concurrent processing, distributed memory, and efficient processor networking.

Parallel processing increases throughput by breaking a problem into subparts and then executing them concurrently. Multiple processors that achieve high levels of performance with high levels of interconnectivity have also been developed in the past few years. Many of these systems use a hypercube pattern,[5] such as the one for this project (manufactured by Topologix, Inc.), which connects 16 INMOS T800 Transputers in the hypercube configuration.

This report describes the implementation of a QR algorithm on the Topologix hypercube architecture. The QR algorithm, which provides a very fast convergence rate, may be used to calculate the eigenstructure for the EMVDR beamformer. The first section of the report provides an overview of the EMVDR algorithm. Following sections describe the QR factorization, an adaptable eigenvector/eigenvalue decomposition using the QR factorization, and the Topologix hypercube architecture and special software routines. In the final section, observations about the implementation of the algorithm are discussed. The source code for the EMVDR beamformer with the adaptable QR algorithm is included as appendix A. Appendix B contains the source code for broadcast and exchange routines.

ENHANCED MINIMUM VARIANCE ALGORITHM

The spatial cross-spectral density matrix (CSDM) at a frequency ω for an N sensor array with K sources is written as

$$C = \sigma_s^2 M\Lambda M^H + \sigma_o^2 I_N ,$$

where M is an N x K matrix of orthonormal eigenvectors of the source subspace, Λ is a diagonal matrix of the eigenvalues associated with the eigenvectors, and I_N is an N x N identity matrix. The values σ_s and σ_o are the spectral densities, respectively, of the source and noise. The enhanced CSDM is defined with the scalar enhancement factor $(e)^4$ as

$$C(e) = e\sigma_s^2 M\Lambda M^H + \sigma_o^2 I_N . \qquad (1)$$

The factor e ($\geq$ 1) makes the source subspace louder than the noise subspace, thus allowing the sources to dominate in the CSDM. Equation (1) can be rewritten as [6]

$$C(e,s) = e \sum_{i=1}^{s} \mu_i \underline{m}_i \underline{m}_i^H + I_N \ , \qquad (2)$$

where

μ_i = i^{th} eigenvalue,

$\underline{m}_i$ = corresponding eigenvector, and

s = dominant number of sources.

Inverting equation (2) yields

$$C(e,s)^{-1} = I_N - \sum_{i=1}^{s} \frac{e\mu_i}{1 + e\mu_i} \underline{m}_i \underline{m}_i^H \ . \qquad (3)$$

The weight vector for steering direction $\underline{d}$ is defined by

$$\underline{w}(e,s) = \frac{C(e,s)^{-1}\,\underline{d}}{\underline{d}^H\, C(e,s)^{-1}\,\underline{d}} \quad . \tag{4}$$

Combining equations (3) and (4), we get the null steerer equation

$$\underline{w}(e,s) = \frac{(I_N - \sum_{i=1}^{s} \frac{e\mu_i}{1 + e\mu_i}\, \underline{m}_i \underline{m}_i^H)\, \underline{d}}{N - \sum_{i=1}^{s} \frac{e\mu_i}{1 + e\mu_i}\, |\underline{m}_i^H\, \underline{d}|^2} \quad . \tag{5}$$

The solution to equation (5) is used in the EMVDR beamformer equation

$$\underline{y} = \underline{w}(e,s)^H\, \underline{x} \ ,$$

where $\underline{x}$ is the frequency domain input data vector. The EMVDR algorithm exploits the fact that threshold signals (eigenvalues near noise variance) do not need to be captured by the estimated CSDM because they are filtered out by the sidelobes. The EMVDR approach requires only high level signals (interferences) to obtain nearly optimum detection performance.[4,7]

QR FACTORIZATION

The QR factorization is the most popular technique for evaluating the complete set of eigenvalues and eigenvectors of a Hermitian or symmetric matrix.[8,9,10] It is used by most numerical analysis subroutines because it offers extremely fast convergence rates. The QR factorization is a matrix realization of the Gram-Schmidt orthonormalization process.[8] The theorem is stated as follows:

If A is a complex n x n matrix, then there is a unitary matrix Q ($Q^H = Q^{-1}$) and an upper triangular matrix R, such that A = QR. If A is nonsingular (its inverse exists), then Q and R can be uniquely determined.

Householder transformations are accumulated to form the Q matrix[3,9]

$$Q = H_1 H_2 H_3 ... H_k \, . \tag{6}$$

Each Householder matrix H_k is calculated according to the form

$$H_k = I - \frac{\underline{u}_k \underline{u}_k^H}{\rho_k} , \tag{7}$$

where

ρ_k = first element in $\underline{u}_k$,

$\underline{u}_k = \underline{e}_k + \frac{\underline{a}}{\sigma \|\underline{a}\|}$,

$\underline{e}_k$ = column k of identity matrix,

$\underline{a}$ = column of original matrix (A),

$\|\underline{a}\|$ = Euclidean norm of column $\underline{a}$, and

σ = ± 1, depending on sign of first component of $\underline{a}$.

Each H_k is chosen such that it will introduce zeros into the last n-k components of the k^{th} column of A. The original matrix is transformed into an upper triangular matrix R by premultiplying A by each H_k. For example, if A originally has the form

$$A = \begin{bmatrix} x & x & x & x \\ \boxed{x} & x & x & x \\ \boxed{x} & x & x & x \\ \boxed{x} & x & x & x \end{bmatrix},$$

then the transform H_1 is determined to clear the last n-1 components in the first column of H_1A. When the transform is applied to A, zeros are introduced into the boxed elements as follows:

$$H_1A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & x & x & x \\ 0 & \boxed{x} & x & x \\ 0 & \boxed{x} & x & x \end{bmatrix},$$

where r_{ij} is an element of the final R matrix (upper triangular). The matrix H_2 is chosen to introduce zeros into the n-2 boxed elements in the second column. The first component of $\underline{u}_2$ that determines H_2 is zero; therefore, the components labeled r_{ij} and 0 are undisturbed by the application of H_2 to H_1A. The reduction is complete when the matrix A is upper triangular. The result is

$$H_4H_3H_2H_1A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}.$$

The basic QR iteration follows:[10] Given an n x n matrix A_0 whose eigenstructure is desired, a unitary matrix Q_k and an upper triangular matrix R_k are found, such that

$$A_k = Q_kR_k. \tag{8}$$

The next matrix in the iterative sequence is formed from the product

$$A_{k+1} = R_kQ_k. \tag{9}$$

The algorithm is described by the compact equation

$$R_kQ_k = Q_{k+1}R_{k+1}.$$

The sequence of matrices A_k will converge to a diagonal or upper triangular matrix with the eigenvalues of the matrix A_0 as diagonal entries arranged in descending order.

The QR factorization is very efficient, but it is not directly applicable to an adaptively updated covariance matrix because each iterative step does not involve the original matrix. In an adaptive environment, the matrix A_0 is estimated by averaging over a sequence of observed samples and therefore varies with time. After each time update, the QR factorization must be performed on a fully dense matrix. It is therefore desirable to adapt the estimates of the eigenstructure (upper triangular matrix) as each new data sample is available.

ADAPTIVE QR ALGORITHM

We can examine the role that the initial matrix A_0 plays in the QR iteration to develop an adaptive version of the QR algorithm.[10] From equation (8), we know that

$$Q_k^H A_k = R_k \ . \tag{10}$$

Using equation (10), we express the QR sequence of equation (9) as

$$A_{k+1} = R_k Q_k = Q_k^H A_k Q_k \ . \tag{11}$$

Repeated application of equation (11) shows the relationship of A_{k+1} to the original matrix A_0 as

$$A_{k+1} = T_k^H A_0 T_k \ , \tag{12}$$

where T_k is the accumulation of Q factors, $T_k = Q_0 Q_1 ... Q_k$. In the limit $k \rightarrow$ infinity, the columns of T converge to the eigenvectors of the original matrix (A_0). When k = K, where K is an arbitrary number greater than zero, the eigenvalues (A_K) and the eigenvectors (T_K) of the original matrix A_0 can be used in the EMVDR null steerer shown in equation (5).

The matrix A_0 varies with time as new data samples become available in an adaptive application such as EMVDR. It can be updated by means of an additive rank-one modification:

$$A_{0,t+1} = (1 - \alpha) A_{0,t} + \alpha \underline{x}_{t+1} \underline{x}_{t+1}^H \ . \tag{13}$$

where

t = update time,
$\underline{x}_{t+1}$ = new data vector from sensors, and
$0 < \alpha < 1$ = constant exponential weighting factor.

When the QR iteration has reached step k = K, the current QR product matrix $A_{K,t}$ can be replaced by the updated matrix $A_{0,t+1}$ by use of equation (12):

$$A_{K,t+1} = T^{H}_{K-1,t}\, A_{0,t+1}\, T_{K-1,t} \quad . \tag{14}$$

The second subscript on T indicates that it now depends on the update time; i.e., $T_{K,t}$ is the transformation from $A_{0,t}$ to $A_{K,t}$.

Use of equation (13) allows the new data update of equation (14) to be written in the form

$$\begin{aligned} \underline{z}_{t+1} &= T^{H}_{k-1,t}\, \underline{x}_{t+1} \\ A_{k,t+1} &= (1 - \alpha)A_{k,t} + \alpha \underline{z}_{t+1}\underline{z}^{H}_{t+1} \, , \end{aligned} \tag{15}$$

which is shown graphically in figure 1. Premultiplying the new data vector $\underline{x}_{t+1}$ by the accumulation matrix $T^{H}_{k-1,t}$ ensures that the next iteration will converge to an upper triangular result that is similar to the original matrix. The complete adaptive QR algorithm uses two simultaneous recursions. The first set of recursions performs the QR iteration:

$$\begin{aligned} A_{k,t} &\rightarrow Q_{k,t}R_{k,t} \\ A_{k+1,t} &= R_{k,t}Q_{k,t} \\ T_{k,t} &= T_{k-1,t}Q_{k,t} \, . \end{aligned} \tag{16}$$

The second recursion is given by equation (15) and is invoked whenever a new data vector becomes available. The entire process is illustrated in figure 2.

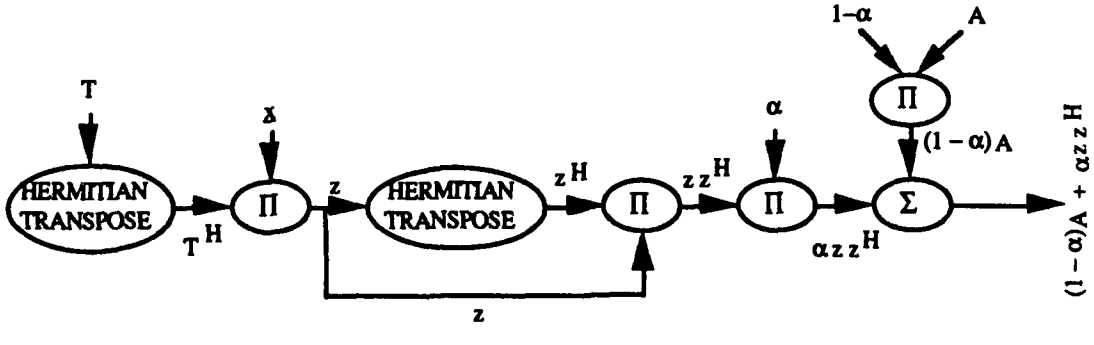

Figure 1. Rank-One Eigenvalue Update

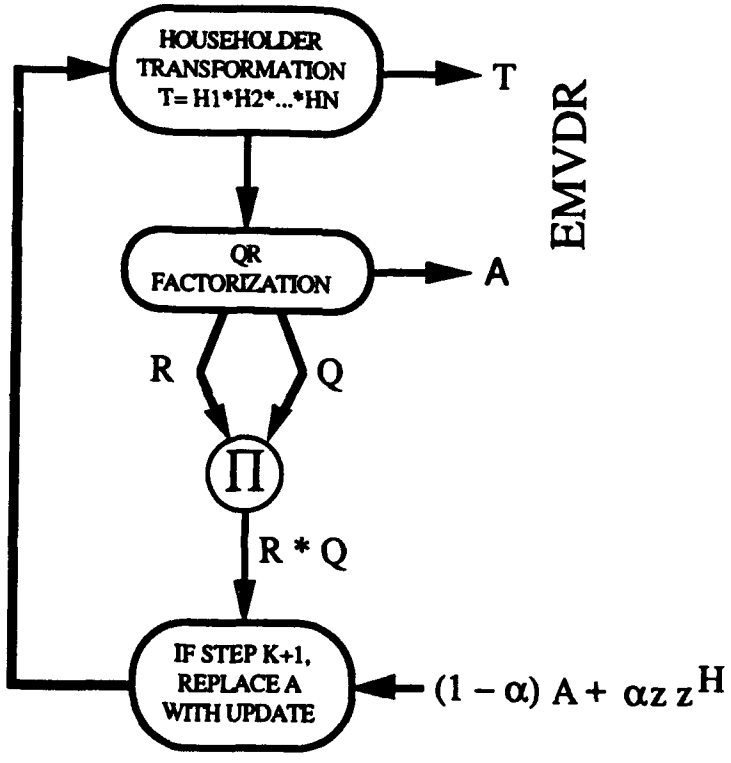

Figure 2. Adaptable QR Algorithm

TOPOLOGIX MULTICOMPUTER SYSTEM

HYPERCUBE ARCHITECTURE

The hypercube architecture was chosen because it has unique qualities that are appropriate to both the QR and EMVDR algorithms:

1. Multiple instructions and multiple data

   The architecture enables all processors to execute similar application programs on different data sets without instruction synchronization. The program and data are stored in local memory, so that computation is within the individual processor. This particular advantage allows each processor to perform the reduced dimensioned EMVDR (using largest eigenvalues/eigenvectors) algorithm for a different beam direction, which produces n weight vectors (for an architecture with n processors) simultaneously.

2. Multiple topologies

   The architecture can emulate other network topologies, such as linear, mesh, torus, or tree, which allows programs to change the topology "on-the-fly" so that functions are performed more efficiently. For example, the architecture could be configured as a mesh (through software) to perform the QR algorithm, but could still use the hypercube network to broadcast global information.

3. Global data passing (with minimal communications)

   Another advantage of the architecture is its small diameter. Any processor can send data to any other in, at most, d steps (for a d-dimensional hypercube). Consider a 13-dimensional hypercube containing 8192 processors. The maximum number of steps for a message to travel between any two processors is 13. Compared with either a linear or mesh configuration, communication is significantly reduced for broadcasting data.

The Topologix is a multicomputer system that can be configured into a hypercube interconnection network. The address of a processor is a binary number of length d. Each

processor has a link to its nearest neighbors. A nearest neighbor is a processor whose address differs by exactly 1 bit (Gray code). The topology guarantees that no two processors are more than d links apart. Table 1 identifies features of the INMOS T800 transputer chip used in the Topologix system. Figure 3 shows the Topologix four-dimensional hypercube configuration and corresponding binary addresses.

Table 1. INMOS T800 Transputer Features

T800 Features
20 MHz clock rate
32 Bit word
4 Communication links
20 Mbits/sec I/O per link
4 Mbytes RAM per processor
14 RISC MIPs (peak) instruction rate, single precision
2.2 MFLOPs (peak) instruction rate, single precision

PROGRAMMING

Converting a problem from a sequential system (single computer) to a multicomputer system is not a straightforward task. On a sequential machine, there is no concern over message passing, synchronization, or load balancing, whereas on a multicomputer, these factors are important.[11] Mapping an algorithm onto a multicomputer includes making decisions dependent on communication and processing operations.[12] For example,

- How should data be distributed across local memories?
- Should each processor repeat identical computations or share results?
- How many processors should be used for the problem?

Algorithms requiring frequent data exchanges may not perform well on some hypercube multicomputers because of low communication bandwidths. Replicating data and operations across processors or decomposing the problems into larger parts can reduce communications.[13]

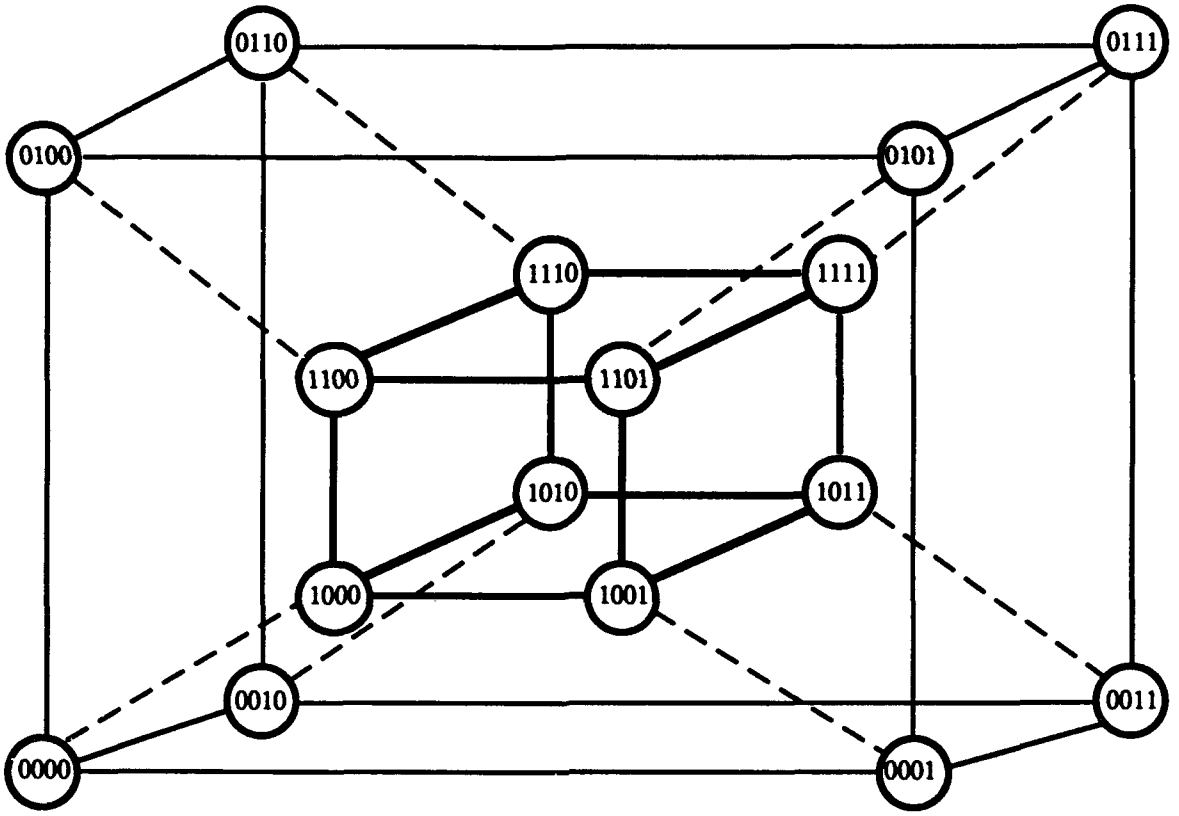

Figure 3. Four-Dimensional Hypercube

HOUSEHOLDER REDUCTION

Communication in the sequential QR algorithm is quite intensive because of the Householder reduction.[9] Columns of the original matrix were mapped onto processors in a wrapped fashion to reduce communications. Wrapping resulted in column *j* residing in processor (*j mod p*), where *p* is the number of processors used for the problem (16 in this case).

Communication is needed to distribute the vector $\underline{u}_k$ to all processors. Sending $\underline{u}_k$ to all processors enables the calculation H_kC to be performed in a parallel fashion. In the sequential version of the reduction operation, equation (7) must be multiplied by the (*n-k*) columns to the right of column *k*. The parallel version performs most of the reduction simultaneously. Every processor takes its turn to generate the vector $\underline{u}_k$ and to broadcast it to every other processor. Each processor holds n/p columns of the upper triangular matrix R when the reduction is complete. An exchange routine (discussed later) assembles the completed R matrix in each processor.

QR FACTORIZATION

The number of multiplications necessary to solve equation (6) is on the order of n^4 operations. It is desirable to perform most of the matrix multiplications simultaneously. To perform the accumulation process of equation (6), the Topologix was configured in a mesh topology through software controls. With a mesh topology, multiple pipelines can perform the process in parallel. Figure 4 illustrates the mesh topology used for the accumulation. Each processor multiplies H_kH_{k+1} (except right boundary processors). Once the accumulation is complete, processor 0 holds the entire Q matrix, which is then sent to all processors. Since each processor holds the entire Q and R matrices, the communication costs of using these results in later steps of the algorithm are reduced.

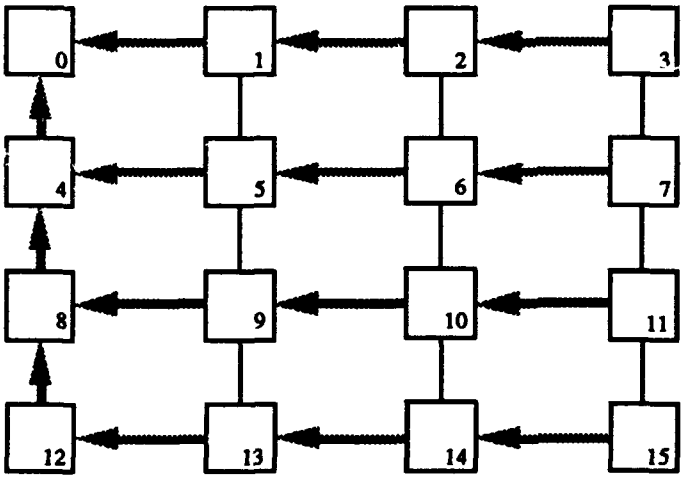

Figure 4. Communication Flow for Q Matrix Accumulation

MATRIX-MATRIX MULTIPLICATION

Matrix multiplication is one of the simplest parallelization problems and has been studied extensively for execution on parallel architectures using various decomposition methods. Decomposition involves dividing a matrix into subunits and assigning each subunit to a processor. A subunit can be a row, column, element, or smaller matrix.

A square sub-block decomposition[14] was performed because the Topologix was already configured in a mesh topology (to perform Q accumulation). The sub-block decomposition did not require communication because each processor already held the R and Q matrices. Each processor performed the following matrix multiply:

$$A^{mk} = \sum_{n} R^{mn} Q^{nk},$$

where A^{mk}, R^{mn}, and Q^{nk} are all square submatrices. Each processor performs $n^3/16$ operations to obtain the submatrix result. The submatrices are then assembled and the resulting matrix is broadcast to all processors.

COMMUNICATION ROUTINES

To perform the QR algorithm, it was necessary to devise efficient ways to pass and exchange data among processors on the Topologix system. Two routines were developed to aid in data movement.

Broadcast

In this scenario, where one processor must efficiently distribute results to all processors, the hypercube configuration solves the problem. In a hypercube, each node has *d* neighbors (d-dimensional) with addresses that differ by exactly 1 bit. The bit positions correspond to port numbers (e.g., 0000 to 0001 over port 0 and 0000 to 0100 over port 2). The broadcast routine uses these bit positions to send data out all ports:

For b = 0 to d-1

1. *Processors are divided into cubes according to the value of bit position b.*
2. *Processors in corresponding positions send/receive data.*

The broadcast routine begins with one processor (2^0) having the correct data. After each iteration, b, the correct data is held in 2^b processors. After four iterations, 16 processors hold the correct data. Table 1 shows the measured execution times for various matrix sizes. The source code for the broadcast routine is included in appendix B.

Exchange

An operation to assemble scattered data and then distribute the result to every processor will reduce overall communications. The exchange routine performs both steps simultaneously using a concept similar to the broadcast routine:

For b = 0 to d-1

1. *Processors are divided into two (d-1) cubes according to value of bit b.*
2. *Processors in corresponding positions exchange data.*
3. *Each processor concatenates its own data with the received data to form new data.*

The exchange routine assumes that the data held in each processor are one or more columns of the resultant matrix. The concatenation is performed by adding the column vectors to a matrix (originally full of zeros). After four iterations, every processor holds the completed matrix. Table 2 shows the measured execution times for various matrix sizes. The source code for the exchange routine is included in appendix B.

Table 2. Measured Execution Times for Broadcast and Exchange Routines

Matrix size	Bytes	Broadcast (sec)	Exchange (sec)
4 x 4	128	.0032	.0102
8 x 8	512	.0034	.0208
16 x 16	2048	.0072	.0498
32 x 32	8192	.0144	.1654

CONCLUSIONS AND RECOMMENDATIONS

The parallel adaptive QR algorithm was run on the Topologix, and the execution times are summarized in table 3. The table shows only two matrix sizes because the Topologix has a communication capacity of 8-kbyte blocks; therefore the QR algorithm (as implemented) could not handle complex matrices larger than 32 x 32. Even with the small size used for the test problem, the results showed improvement over the best sequential method. The sequential method was executed on a single Topologix processor.

Table 3. Measured Execution Times for QR Algorithms

Matrix size	Processors	Parallel (sec)	Sequential (sec)
16 x 16	16	1.24	1.21
32 x 32	16	9.12	15.11

Table 4 shows the order of operations for the parallel and sequential QR algorithms (including two matrix multiplies). Estimates of larger matrix decompositions were not made because the cause of the overhead time (measured - ideal) was unknown. The table shows that the computation grows faster than the communication for increasing n. However, because there are many other sources of overhead that also appear when larger problems are distributed over concurrently operating processors,[13] estimating the execution time of these problems becomes difficult. One major source of overhead is processor communication. Communication between Topologix processors is very inefficient, as shown in figure 5, which depicts measured data rates versus number of bytes. The data channels are rated at 20 million bits per second (Mbps). The channel is only 31 percent utilized with the maximum amount of data (8 kbytes). The poor channel utilization is related to the Topologix operating system and is not inherent to hypercube architectures.

The speedup of an ideal algorithm implementation would be p (number of processors used), with each processor performing 1/p operations of the total problem (load balanced). Figure 6 is a graphical representation of table 4. It shows the ideal speedup that could be achieved with

the QR algorithm. The speedup curve in the figure is ideal because it does not include any overhead (i.e., interprocessor communication, memory accesses, or synchronization).

Table 4. Order of Operations

Algorithm Type	Calculations	Communications
Sequential	$2n^4$	--
Parallel	$\frac{n^4 + 45n^3}{8}$	5n

The QR implementation described is not load balanced because of the Householder reduction; there are not enough matrix elements, after the reduction, to spread over the processors. Fox[14] proposes a scattered decomposition for performing the Householder reduction to preserve load balance. The scatter method, however, adds more communication, $O(n^2)$, which can be devastating for a high overhead system like the Topologix.

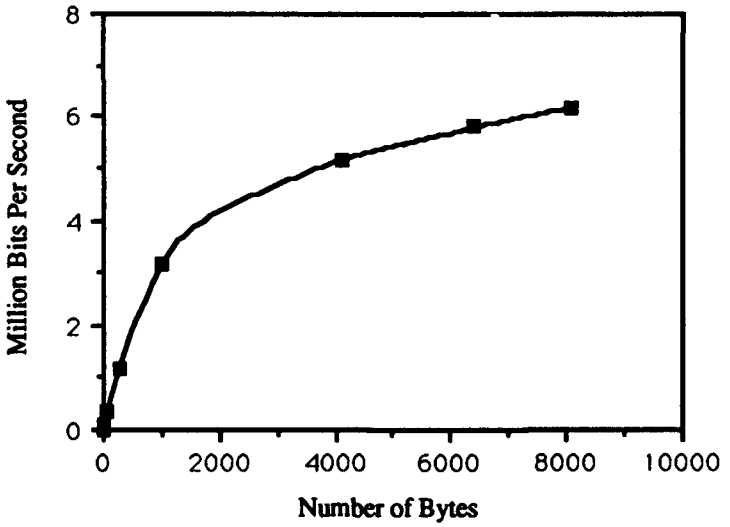

Figure 5. Measured Data Transfer Rates

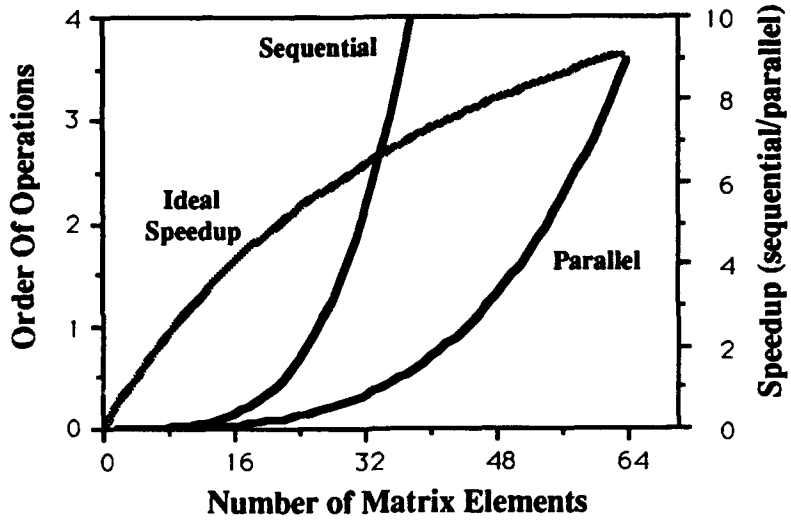

Figure 6. Order Of Operations and Ideal Speedup

The QR factorization uses eigenstructure techniques that need at least an order of magnitude more computations than the calculation of the adaptive weights. Therefore, concentrating on the eigenstructure techniques is one approach to a parallel implementation of the EMVDR application. The QR decomposition (preceded by Householder reduction) described above does not provide load balancing. In reference 15, the QR algorithm is replaced by an iterative sequence that converges to the source subspace eigenstructure. This method requires many iterations and appears to be difficult to program onto a hypercube architecture efficiently; however, research into other eigenstructure methods for hypercube architectures is proceeding[16] and should be studied for use in the EMVDR application.

The hypercube architecture can solve linear algebra algorithms efficiently (such as those used in signal processing applications) as well as provide modular growth for future computing needs. The continuing study of these architectures should focus on such systems as the Connection Machine and the NCUBE 6400 Series. These systems use faster processing engines, faster communication paths, and different operating systems than the Topologix system.

Future research should also concentrate on the relationship between parallel architectures and eigenstructure methods. Eigenstructure algorithms should be tailored to architectures to exploit hardware assets (e.g., processing power, distributed memory, or networking ability). Ongoing

work in linear algebra algorithms (including eigenstructure techniques) on various architectures in the academic and industrial communities should continue to be supported by the Navy.

REFERENCES

1. N. L. Owsley, "Systolic Array Adaptive Beamforming," NUSC Technical Report 7981, Naval Underwater Systems Center, New London, CT, 21 September 1987.

2. M. H. Leonhardt, "Implementation of Minimum Variance Distortionless Response (MVDR) Adaptive Beamforming Algorithm," NUSC Technical Document 8543, Naval Underwater Systems Center, New London, CT, 19 July 1989.

3. G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore, MD, 1983.

4. N. L. Owsley, "Enhanced Minimum Variance Beamforming," NUSC Technical Report 8305, Naval Underwater Systems Center, New London, CT, 18 November 1988.

5. J. P. Hayes, "Hypercube Supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, 12 December 1989, pp. 1829-1841.

6. W. Beyer, *CRC Standard Mathematical Tables*, CRC Press, Boca Raton, FL, 1981.

7. S. Haykin, ed., *Array Signal Processing*, Prentice Hall , Englewood Cliffs, NJ, 1985, Chapt. 3.

8. D. S. Watkins, "Understanding the QR Algorithm," *SIAM Review*, vol. 24, no. 4, October 1982, pp. 427-440.

9. J. J. Dongarra et al., *LINPACK User's Guide*, SIAM, Philadelphia, 1979, Chapt. 9.

10. K. C. Sharman, "Adaptive Algorithms for Estimating the Complete Covariance Eigenstructure," *Proceedings of the ICASSP 86*, 1986, pp. 1401-1404.

11. D. Sweetman and J. Muñoz, "Measures Of Effectiveness (MOEs) For Parallel Architectures," *Proceedings of the IEEE Conference on Systems, MAN, and Cybernetics*, vol. II, 1989, pp. 630-635.

12. S. Ranka, Y. Won, and S. Sahni, "Programming a Hypercube Multicomputer," *IEEE Software*, September 1988, pp. 69-77.

13. A. Frey, "Hypercube Architectures and Their Application to Signal Processing," *Proceedings of SPIE: Highly Parallel Signal Processing Architectures*, vol. 614, 1986, pp. 74-81.

14. G. Fox et al., Solving Problems on Concurrent Processors, vol. I, Prentice Hall, Englewood Cliffs, NJ, 1988.

15. M. Kaveh and J. Yang, "Adaptive Eigensubspace Algorithms for Direction or Frequency Estimation and Tracking," *IEEE Transactions on ASSP*, vol. 36, no. 2, February 1988, pp. 241-251.

16. I. Ipsen and E. Jessup, "Two Methods for Solving the Symmetric Tridiagonal Eigenvalue Problem on the Hypercube," *Hypercube Multiprocessors*, SIAM, 1987.

APPENDIX A

SOURCE CODE FOR EMVDR APPLICATION

```
/*****************************************************************************
    EMVDR IMPLEMENTATION ON THE TOPOLOGIX HYPERCUBE

                        AUTHOR:  Manfred Leonhardt

    THIS PROGRAM PERFORMS THREE MAIN FUNCTIONS:

    1)  GENERATES RANDOM GAUSSIAN DATA TO BE USED AS SENSOR
        OUTPUT,

    2)  PARALLEL: QR DECOMPOSITION, RQ MULTIPLY, AND TQ
        MULTIPLY TO GET EIGENVALUES AND EIGENVECTORS, AND

    3)  ENHANCED MINIMUM VARIANCE DISTORTIONLESS RESPONSE
        ADAPTIVE BEAMFORMING AS SPECIFIED IN NUSC TR 8305
        NOVEMBER 1988.

      THIS PROGRAM EXECUTES ON THE TOPOLOGIX  SYSTEM WITH 16
      PROCESSING NODES (4-DIM HYPERCUBE).  IT IS LOADED WITH
     THE COMMAND:
                              loadgo a15-0 -w 2500000 EMV
*****************************************************************************/
#include <logixos/events.h>    /* TOPOLOGIX EVENTS FOR MESSAGE PASSING */
#include <logixos/net.h>         /* MESSAGE PASSING NETWORK MESSAGES */
#include <stdio.h>                     /* OUTPUT VARIABLES AND FLAGS */
#include <math.h>                   /* MATH FORMULAS ASSIGNED VALUES */
#include <sys/time.h>                  /* TIME OF DAY FOR RANDOM SEED */
#include "complex.h"           /* COMPLEX NUMBERS AND MATH ROUTINES */
#include "matrix.h"                 /* MATRICES AND VARIOUS ROUTINES */
#include "QR.h"          /* ROUTINES FOR THE HOUSEHOLDER ALGORITHM */
#include "QRparallel.h"        /* ROUTINES & STRUCTURES FOR TOPOLOGIX */
#include "bcast.h"       /* BROADCAST AND COLLECT DATA ON TOPOLOGIX */

/*    DEFINITIONS THAT ARE NEEDED FOR THE EMVDR ALGORITHM      */
#define freq          100.
#define wavelength    1500. / freq
#define spacing       wavelength / 2.
#define degrad        *M_PI / 180.        /*  DEGREE TO RADIAN CONVERSION  */
#define p       16    /*  NUMBER OF PROCS PROGRAM WILL EXECUTE ON  */
#define prcs    4       /*  NUMBER OF ROWS FOR PROCS: SQRT (P) ABOVE  */

enum boolean {TRUE, FALSE};

/* ************************************************************************** */

Calculate_H (Umat, col, Hmat, n)
 complex_matrix       Umat, Hmat;
 int                  col, n;
/*  ------------------------------------------------------------------------
   THIS ROUTINE WILL GENERATE THE H MATRIX NEEDED TO
   CALCULATE THE Q MATRIX TO SOLVE THE QR DECOMPOSITION.

     INPUT:
             Umat :   type complex_matrix
```

```
                    Contains the columns that will be multiplied
                    to form the H matrix.

              col :  type integer
                    Column of interest in the U matrix.

                n :  type integer
                    Number of columns/rows in the H matrix.

    OUTPUT:
            Hmat :  type complex_matrix
                    Contains the result of: I-(U*Uh)/U(1,1).

    THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

                            conj, cmult, and cdiv.
-------------------------------------------------------------------------------- */
{
    complex   tempc;
    int       i, j;

    /* ---------------------- PERFORM I - (U * Uh / Uii) ---------------------- */
    if (col != n-1)
    {
    for (i = col; i < n; ++i)
        for (j = col; j < n; ++j)
        {
            tempc = conj (Umat[j][col]);
            Hmat[i][j] = cmult (Umat[i][col], tempc);
            Hmat[i][j] = cdiv (Hmat[i][j], Umat[col][col]);
            /* --- SUBTRACT RESULT FROM THE IDENTITY MATRIX --- */
            Hmat[i][j].im = -Hmat[i][j].im;
            if (i == j)
                Hmat[i][i].re = 1. - Hmat[i][i].re;
            else
                Hmat[i][j].re = -Hmat[i][j].re;
        }
    }
    /* --- FILL IN OTHER PART OF H MATRIX TO IDENTITY MATRIX. --- */
    for (i = 0; i < col; ++i)
        for (j = 0; j < n; ++j)
        {
            Hmat[i][j].im = 0.;
            Hmat[j][i].im = 0.;
            if (i == j)
                Hmat[i][i].re = 1.;
            else
            {
                Hmat[i][j].re = 0.;
                Hmat[j][i].re = 0.;
            }
        }
}
```

```
/* ************************************************************************** */

Parallel_Matrix_Mult (first_matrix, second_matrix, result, rows, proc_numbers)
 complex_matrix      first_matrix, second_matrix, result;
 int                 proc_numbers;
/*  ------------------------------------------------------------------------------
     THIS ROUTINE WILL PERFORM MATRIX-MATRIX MULTIPLICATION
     IN PARALLEL BY USING MULTIPLE PROCESSORS ON THE
     TOPOLOGIX SYSTEM. THE TOTAL MATRIX IS ASSUMED TO BE IN
     EVERY PROCESSOR. EACH PROCESSOR WILL WORK ON A SQUARE
     SUBBLOCK (size: # of rows in matrix / # of processors in row) OF THE
     ORIGINAL MATRIX.

     INPUT:
          first_matrix  :    type complex_matrix
                             Upper triangular matrix from QR decomposition.

         second_matrix  :    type complex_matrix
                             Normalized matrix from QR decomposition.

                 rows   :    type integer
                             Number of rows that are in the Q or R matrices.

          proc_numbers  :    type integer
                             Number of rows or columns of processors (sqrt(p)).

     OUTPUT:
                 result :    type complex_matrix
                             Diagonal contains the eigenvalues of the matrix
                             that was decomposed by the QR decomposition.

     THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

                          Meshid, cmult, and cadd.
  ------------------------------------------------------------------------------ */
{
     int         Ablockcol, Ablockrow, Bblockcol, Bblockrow;
     int         icol, irow, i1, i2, j1, j2, count, sp;
     complex     tempc;

     Meshid (&irow, &icol);
     Ablockrow = irow;
     Ablockcol = irow;
     Bblockrow = irow;
     Bblockcol = icol;
     sp = rows / proc_numbers;
     for (count = 0; count < proc_numbers; ++count)
     {
       for (i1 = Ablockrow*sp; i1 < (Ablockrow*sp)+sp; ++i1)
         for (j2 = Bblockcol*sp; j2 < (Bblockcol*sp)+sp; ++j2)
               for (i2=Bblockrow*sp,j1=Ablockcol*sp;i2<(Bblockrow*sp)+sp; ++i2,++j1)
               {
                     tempc = cmult (first_matrix[i1][j1], second_matrix[i2][j2]);
                     result[i1][j2] = cadd (tempc, result[i1][j2]);
```

```
            }
        ++Ablockcol;
        if (Ablockcol >= proc_numbers) Ablockcol = 0.;
        ++Bblockrow;
        if (Bblockrow >= proc_numbers) Bblockrow = 0.;
    }
}

/* ************************************************************************** */

QR_Iteration (Cmat, Tmat, n)
complex_matrix      Cmat, Tmat;
int                 n;
/*  ---------------------------------------------------------------------------
    THIS ROUTINE WILL PERFORM A QR DECOMPOSITION USING p
    PROCESSORS IN A PARALLEL FASHION.  THE MATRIX IS
    BROADCAST TO ALL PROCESSORS WITH EACH WORKING ON A
    SMALL PART OF THE MATRIX.  WHEN THE ROUTINE IS COMPLETE
    ALL THE PROCESSORS WILL CONTAIN THE Q, R, T (accumulation of
    Qs), and Eigs (eigenvalues of matrix). THIS ROUTINE WAS TAKEN FROM
    THE  LINPACK USER'S GUIDE, CHAPTER 9.

    INPUT:
            Cmat :  type complex_matrix Current covariance matrix

            Tmat :  type complex_matrix
                    Identity matrix first time through and accumulation of Qs
                    for every other time.

               n :  type integer
                    Number of rows in the Cmat matrix.

    OUTPUT:
            Cmat :  type complex_matrix
                    Matrix with diagonal values being the eigenvalues
                    of the covariance matrix.

            Tmat :  type complex_matrix
                    Matrix containing the accumulation with the new Q
                    matrix. The eigenvectors of the covariance matrix.

    THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

        Broadcast, Calculate_H, cassign, cdiv, cmat_equate, cmat_mult,
        cmult, Exchange, Parallel_Matrix_Mult, receive, send, zaxpy, zdot,
        znrm2, and zscal
--------------------------------------------------------------------------- */
{
    int             k, l, i, j, loopstart, xcount;
    int             iam, usize, prow, pcol, ihave;
    float           temp, Enrm;
    complex         sigma, nrmc, tempc, cnst, t;
    complex_matrix  Rmat, Umat, Hmat, H1mat, H2mat, Tempmat;
    FILE            *data, *fopen ();
```

```
usize = sizeof(Umat);                    /* GET SIZE FOR MESSAGE PASSING */

/* ----- GET THE VIRTUAL NODE ID FOR THE COMPUTATIONS ------ */
iam = v16_node_number[getnodeid()];

/* ------ TRANSFER THE COVARIANCE MATRIX FOR ROUTINE ------- */
cmat_equate (Rmat, Cmat, n);
/* ------ SEND OUT THE ENTIRE MATRIX TO ALL PROCESSORS ------ */
Broadcast (Rmat, usize, 0);
/* ---- START THE MAIN COMPUTATIONS FOR HOUSEHOLDERS ---- */
for (l = 0; l < n-1; ++l)
{
    k = l % p;   /* WHAT PROCESSOR HOLDS THE NEEDED COLUMN ? */

    /* --- GET HYPERCUBE NODE ID FOR BROADCAST ROUTINE --- */
    ihave = v16_to_hypercube_number[k];

    if (iam == k)
    {    /* WHAT PROCESSOR NEEDS TO CALCULATE U COLUMN ? */
        Enrm = znrm2 (n-l, Rmat, l, n);          /* NORM OF COLUMN L */
        if (Enrm != 0)
        {
            if (Rmat[l][l].re != 0.)
            {
            /* ------ GET SIGN FOR U VECTOR CALCULATION ------ */
                temp = cabs (Rmat[l][l]);
                sigma.re = Rmat[l][l].re/temp;
                sigma.im =  Rmat[l][l].im/temp;
            }
            else
                sigma = cassign (1., 0.);
            nrmc = cassign (sigma.re*Enrm, sigma.im*Enrm);
            tempc = cassign (1., 0.);
            cnst = cdiv (tempc, nrmc);
            /* ---------------- CONSTRUCT U VECTOR ---------------- */
            zscal (n-l, cnst, Rmat, l, n, Umat);
            Umat[l][l].re = Umat[l][l].re + 1.;
            Rmat[l][l] = cassign (-nrmc.re, -nrmc.im);
            for (i = l+1; i < n; ++i)
                Rmat[i][l] = cassign (0., 0.);
            for (i = 0; i < l; ++i)
                Umat[i][l] = cassign (0., 0.);
        }
    }

    /* ---------- SEND OUT U MATRIX TO ALL PROCESSORS ---------- */
    Broadcast (Umat, usize, ihave);

    if (iam == k)
    {                        /* PROCESSOR THAT CALCULATED U VECTOR */
        if (Enrm != 0)
        {    /* FINISH CALCULATING OTHER COLS OF R MATRIX */
            for (j = l+p; j < n; j += p)
```

```
                {
                    t = zdot (n-l, Umat, l, Rmat, j, n);
                    t = cdiv (t, Umat[l][l]);
                    tempc.re = -1;
                    tempc.im = 0.;
                    t = cmult (tempc, t);
                    zaxpy (n-l, t, Umat, l, Kmat, j, n);
                }
            }
        }
        else                                    /*  PROCESSOR IS NOT EQUAL TO K  */
        {
            loopstart = iam;
            xcount = 1;
            while (loopstart < l)
            {
                loopstart = (xcount * p) + iam;
                ++xcount;
            }
            for (j = loopstart; j < n; j += p)
            {
                t = zdot (n-l, Umat, l, Rmat, j, n);
                t = cdiv (t, Umat[l][l]);
                tempc.re = -1.;                 /* CREATE NEGATIVE # */
                tempc.im = 0.;
                t = cmult (tempc, t);                       /* NEGATE t */
                zaxpy (n-l, t, Umat, l, Rmat, j, n);
            }
        } /* ------------------- END OF THE IF ELSE LOOP ------------------ */
    } /* ------------------------ END OF THE I LOOP ------------------------ */

    /* - IF PROCESSOR ZERO, EQUATE MATRIX Tempmat TO IDENTITY - */
    if (iam == 0)
    {
        for (i = 0; i < n; ++i)
          for (j = 0; j < n; ++j)
          {
            Tempmat[i][j].im = 0.;
            if (i == j)
                Tempmat[i][i].re = 1.;
            else
                Tempmat[i][j].re = 0.;
          }
    }

    /* --- COLLECT H MATRICES & GENERATE Q MATRIX IN NODE 0 --- */
    for (l = n-p+iam; l >= iam; l -= p)
    {
    /* -------------------- CALCULATE YOUR H MATRIX -------------------- */
        Calculate_H (Umat, l, Hmat, n);
        if (receive (R, usize, H1mat, iam, 0) != -1)
        {
        /* - CREATE TEMPORARY MATRIX MULTIPLICATION HOLDER --
*/
```

```
            for (i = 0; i < n; ++i)
              for (j = 0; j < n; ++j)
              {
                    H2mat[i][j].im = 0.;
                    if (i == j)
                          H2mat[i][i].re = 1.;
                    else
                          H2mat[i][j].re = 0.;
              }
            /* ---------- MULTIPLY THE H's TOGETHER ---------- */
            cmat_mult (l, n, Hmat, H1mat, H2mat);
            /* ---- SWAP THE MATRICES FOR THE NEXT MULTIPLY ---- */
            cmat_equate (Hmat, H2mat, n);
      }
      /* ----- SEND OUT THE MULTIPLICATION RESULT TO LEFT ----- */
      send (L, usize, Hmat, iam, 0);

      if ((iam % 4) == 0)
      {           /*  IF YOU'RE IN COLUMN ZERO THEN PASS DATA UP  */
            if (receive (D, usize, H1mat, iam, 0) != -1)
            {
                  cmat_mult (l, n, Hmat, H1mat, H2mat);
                  cmat_equate (Hmat, H2mat, n);
            }
            send (U, usize, Hmat, iam, 0);
            if (iam == 0)
            {
                  cmat_mult (l, n, Hmat, Tempmat, H2mat);
                  cmat_equate (Tempmat, H2mat, n);
            }
      }
}
/* ---------- CLEAR THE UNIMPORTANT COLUMNS ---------- */
for (l = 0; l < n; ++l)
{
      if (l < p)
      {
            if (l != iam)
            {
                  for (i = 0; i < n; ++i)
                    Rmat[i][l] = cassign (0.,0.);
            }
      }
      else
      {
            if ((l % (l/p)*p) != iam)
            {
                  for (i = 0; i < n; ++i)
                    Rmat[i][l] = cassign (0.,0.);
            }

      }
}
```

```
    /* ------ GET THE ENTIRE R MATRIX INTO EVERY PROCESSOR ------ */
    Exchange (Rmat, usize, n);

    /* -- BROADCAST THE ENTIRE Q MATRIX TO EVERY PROCESSOR -- */
    Broadcast (Hmat, usize, 0);

    /* ------------------ CLEAR THE EIGENVALUE MATRIX ------------------ */
    for (i = 0; i < n; ++i)
      for (j = 0; j < n; ++j)
        Cmat[i][j] = cassign (0., 0.);

    /* ---------------- PERFORM THE RQ MULTIPLICATION ---------------- */
    Parallel_Matrix_Mult (Rmat, Hmat, Cmat, n, prcs);

    /* ---------- PUT EIGENVALUES INTO EVERY PROCESSOR ---------- */
    Exchange (Cmat, usize, n);

    /* --------------------- CLEAR TEMPORARY MATRIX --------------------- */
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            Tempmat[i][j] = cassign (0., 0.);

    /*  ACCUMULATE TRANSFORMATION MATRICES (EIGENVECTORS)  */
    Parallel_Matrix_Mult (Tmat, Hmat, Tempmat, n, prcs);

    /* ---------- PUT EIGENVECTORS INTO EVERY PROCESSOR ---------- */
    Exchange (Tempmat, usize, n);
    cmat_equate (Tmat, Tempmat, n);
}

/* ************************************************************************** */

Update_data (eigvals, trans, inputs, alpha, size)
 complex_matrix     eigvals, trans;
 complex_vector     inputs;                          /* Input vector */
 float              alpha;        /* Forgetting factor for the covariance update */
 int                size;              /* Number of rows or columns in trans */
/*  ------------------------------------------------------------------------
    THIS ROUTINE WILL UPDATE A COVARIANCE MATRIX AND THE
    COMPLEX INPUTS THAT HAVE PREVIOUSLY BEEN GENERATED.

    INPUT:
          eigvals :  type complex_matrix
                     Originally the covariance matrix that is complex
                     Hermitian.  Later it is the current eigenvalue estimation
                     matrix (eigenvalues on diagonal).

           trans :   type complex_matrix
                     Eigenvectors of the covariance matrix to be used to update
                     the new inputs.

          inputs :   type complex_vector
                     New inputs that the hydrophones have received.
```

```
            alpha :   type float
                      Forgetting factor for the covariance update equation.

             size :   type integer
                      Number of rows in the input vector.

     OUTPUT:
           eigvals :  type complex_matrix
                      New updated covariance matrix eigenvalue estimation,
                      includes new data information.

           inputs :   type complex_vector
                      New data that has been updated to conform to the last
                      eigenvectors calculated.

     THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

          cmat_add, cmat_trans, cmatvect_mult, cms_mult, and cvect_equate.
-------------------------------------------------------------------------------- */
{
     int                 i, j;
     complex             thold;
     complex_matrix      transh, result, ans1, ans2;
     complex_vector      updata;

     /* - GET HERMITIAN TRANSPOSE OF TRANSFORMATION MATRIX - */
     cmat_trans (trans, transh, size);
     /* ------------------------ UPDATE INPUT VECTOR ------------------------ */
     cmatvect_mult (transh, inputs, updata, size);
     cvect_equate (updata, inputs, size);

     /* ----- UPDATE COVARIANCE MATRIX USING ABOVE RESULT ------ */
     for (i = 0; i < size; ++i)
          for (j = 0; j < size; ++j)
          {
               thold = conj (updata[j]);
               result[i][j] = cmult (updata[i], thold);
          }
     cms_mult (result, alpha, ans1, size);
     cms_mult (eigvals, 1.-alpha, ans2, size);
     cmat_add (ans1, ans2, eigvals, size);
}

/* ************************************************************************** */

Normal (number1, mean, sd, outreals)
 float       mean, sd;
 float       outreals[];
 int         number1;
/*   ------------------------------------------------------------------------
     THIS ROUTINE WILL GENERATE AN ARRAY OF RANDOM NUMBERS
     WITH A GAUSSIAN DISTRIBUTION.

     INPUT
```

```
            number1 :   type integer
                        Number of values to be in the resulting array.

               mean :   type float
                        Average of the values to be generated.

                 sd :   type float
                        Standard deviation of the data to be generated.

     OUTPUT
            outreals :  type float (vector)
                        Will contain values with Gaussian distribution.
-------------------------------------------------------------------------------- */
{
#define to31 2147483648.0
#define ix   71365
#define FRAND (float)((float)random()/(float)MAXLONG)

     long int    iu;
     double      sqrt (), log (), erand48 ();
     float       root, unif1, unif2, temp, gauss1, gauss2;
     int         i;

     iu = FRAND;
     for (i = 0; i < number1; i+=2)
     {
          do
          {
               iu = ix * iu + 1;          /* FILL UP ARRAY WITH RANDOM NOISE */
               unif1 = iu / to31;
               unif2 = 2. * FRAND - 1.;
               temp = unif1 * unif1 + unif2 * unif2;
          } while (temp > 1.);
          root = sqrt (-2.*log (temp) / temp);
          gauss1 = unif1 * root;
          gauss2 = unif2 * root;
          outreals[i] = mean + sd * gauss1;
          outreals[i+1] = mean + sd * gauss2;
     }
}

/* ********************************************************************************* */

SteerGen (arraysize, MRA, steer)
 int                 arraysize;
 float               MRA;
 complex_vector      steer;
/*   ------------------------------------------------------------------------------
     THIS ROUTINE GENERATES THE STEERING VECTOR FOR THE LOOK
     DIRECTION SPECIFIED.

     INPUT:
          arraysize :   type integer
                        Number of hydrophones in the array.
```

```
            MRA :   type float
                    Bearing of the look direction (degrees).

    OUTPUT:
            steer : type complex vector
                    Will contain the vector for the look direction.

    THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

                    cassign.
-------------------------------------------------------------------------------- */
{
    int         i;
    float       ang, angle;
    double      cos(), sin();

    ang = sin (MRA*M_PI/180.);
    for (i = 0; i < arraysize; ++i)
    {
        angle = i * ang * M_PI;
        steer[i] = cassign (cos(angle), -sin(angle));
    }
}

/* ************************************************************************** */

ProcessInputs (number_of_sensors, how_many_sources, signal_db_values, signal_bearings,
            time_interval, sensor_values)
int                 time_interval, how_many_sources, number_of_sensors;
float               signal_db_values[], signal_bearings[];
complex_vector      sensor_values;
/*  ----------------------------------------------------------------------------
    THIS ROUTINE WILL PROVIDE SENSOR OUTPUTS BY USING THE
    SENSOR INPUTS THAT HAVE BEEN PREVIOUSLY GENERATED.

    INPUT:
    number_of_sensors  :  type integer
                          Contains number of hydrophones per beam.

    how_many_sources   :  type integer
                          Specifies number of sources present to hydrophones.

      signal_db_values :  type float (array)
                          Contains decibel values assigned to each source.

        signal_bearings :  type float (array)
                          Contains the bearing for each signal present.

           time_interval :  type integer
                          Contains time interval in question. Used for sigvar
                          and noise indices.

    OUTPUT:
```

sensor_values : type complex_vector
Will contain hydrophone outputs that were generated.

THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

cadd, cassign, cmult, Normal, and SteerGen.

```
---------------------------------------------------------------------------- */
{
    int                 i, j, k;
    complex             signal[20][400], noise[400], temp;
    float               nreal[400], nimag[400], sreal[400], simag[400];
    complex_vector      steervect, temp2;
    float               sigma;
    double              pow (), sqrt ();
    static enum boolean first = TRUE;

    for (i = 0; i < how_many_sources && first == TRUE; ++i)
    {
    /*  GENERATE SIGNAL POWERS ASSUMING NOISE VARIANCE = 1  */
        sigma = pow (10., signal_db_values[i]/20.) / sqrt(2.);
        Normal (400, 0., sigma, sreal);
        Normal (400, 0., sigma, simag);
        for (k = 0; k < 400; ++k)
            signal[i][k] = cassign (sreal[k], simag[k]);
    }
    first = FALSE;
    /* ---------- GENERATE THE NOISE FOR THE HYDROPHONES --------- */
    Normal (number_of_sensors, 0., 1/sqrt(2.), nreal);
    Normal (number_of_sensors, 0., 1/sqrt(2.), nimag);
    for (k = 0; k < number_of_sensors; ++k)
        noise[k] = cassign (nreal[k], nimag[k]);
    /* ------------ USE VALUES TO GENERATE INPUT VECTOR ------------ */
    for (k = 0; k < number_of_sensors; ++k)
        temp2[k] = cassign (0., 0.);
    for (j = 0; j < how_many_sources; ++j)
    {
    /* - GENERATE STEERING VECTOR FOR EACH SOURCE BEARING -- */
        SteerGen (number_of_sensors, signal_bearings[j], steervect);
        for (k = 0; k < number_of_sensors; ++k)
        {
            temp = cmult (signal[k][time_interval], steervect[k]);
            temp2[k] = cadd (temp2[k], temp);
        }
    }
    /* --------- COMPLETE THE GENERATION OF INPUT VECTOR --------- */
    for (i = 0; i < number_of_sensors; ++i)
        sensor_values[i] = cadd (noise[i], temp2[i]);
}

/* ************************************************************************** */

Inputvalues (file, howmany, SINR, bearings)
 float      SINR[], bearings[];
 FILE       *file;
```

```
int       howmany;
/*  ------------------------------------------------------------------------
    THIS ROUTINE WILL READ IN DECIBEL VALUES AND BEARINGS OF
    ANY SIGNALS THAT ARE PRESENT IN THE SYSTEM.

    THE FILE SHOULD HAVE THE FOLLOWING FORMAT:

            dB_value   space   angle_value

    WHERE THE angle_value IS BETWEEN -90 DEGREES AND 90 DEGREES.

    INPUT:
          file :  type FILE
                  Pointer to the input file.  Must be opened before this
                  routine is called.

     howmany :  type integer
                Number of signals that are present to the system.

    OUTPUT:
          SINR :  type float (vector)
                  Decibel values for all signals present.

      bearings :  type float (vector)
                  Angle locations for all the present signals.
------------------------------------------------------------------------ */
{
    int   i;

    for (i = 0; i < howmany; ++i)
        fscanf (file, "%f %f", &SINR[i], &bearings[i]);
}

/* ************************************************************************ */

EMV (dominant_values, efactor, arraysize, eigvects, eigvals, steer, weights)
int               dominant_values, arraysize;
float             efactor;
complex_vector    steer, weights;
complex_matrix    eigvects, eigvals;
/*  ------------------------------------------------------------------------
    THIS ROUTINE WILL GENERATE THE WEIGHTS TO MINIMIZE THE
    OUTPUT RESPONSE OF A BEAMFORMER BY USING THE
    EIGENVECTORS AND EIGENVALUES OF THE COVARIANCE MATRIX.
    THE FORMULA IS TAKEN FROM NUSC TR 8305, N. OWSLEY,
    NOVEMBER 1988.

    INPUT:
    dominant_values :  type integer
                       Number of dominant signals present.

            efactor :  type float
                       Enhancement factor for EMV usually >= 1.
```

```
            arraysize :  type integer
                         Number of hydrophones in the array.

             eigvects :  type complex matrix
                         Eigenvectors of the covariance matrix.

              eigvals :  type complex matrix
                         Eigenvalues of the covariance matrix.

                steer :  type complex vector
                         Steering vector for look direction.

    OUTPUT:
            weights :  type complex vector
                       Will contain weights to minimize beam response.

    THIS ROUTINE USES THE FOLLOWING EXTERNAL ROUTINES:

        cabs, cadd, cassign, cmat_add, cmatvect_mult, cms_mult,
                       cmult, and conj.
-------------------------------------------------------------------------------- */
{
    int                 i, j, k;
    float               temp, part5, denom, scalar;
    complex             tc, part3, part4;
    complex_matrix      part, part2, old, numer;

    cmat_init (old, arraysize);
    denom = 0.;

    for (i = 0; i < dominant_values; ++i)
    {
        /* ------------- GET SCALAR MULTIPLIER    eu/(1+eu) ------------- */
        temp = efactor * (eigvals[i][i].re - 1.);
        scalar = temp / (1. + temp);
        /* ------------------ SOLVE NUMERATOR PARTIAL ------------------ */
        for (j = 0; j < arraysize; ++j)
            for (k = 0; k < arraysize; ++k)
            {
                tc = conj (eigvects[k][i]);
                part[j][k] = cmult (eigvects[j][i], tc);
            }
        cms_mult (part, -scalar, part2, arraysize);
        cmat_add (part2, old, old, arraysize);
        /* ---------------- SOLVE DENOMINATOR PARTIAL ---------------- */
        part4 = cassign (0., 0.);
        for (j = 0; j < arraysize; ++j)
        {
            tc = conj (eigvects[j][i]);
            part3 = cmult (tc, steer[j]);
            part4 = cadd (part4, part3);
        }
        part5 = cabs (part4) * cabs (part4);
        part5 *= scalar;
```

```
            denom += part5;
        }
        denom = arraysize - denom;

        /* ------------ COMPLETE NUMERATOR COMPUTATIONS ------------ */
        for (i = 0; i < arraysize; ++i)
            for (j = 0; j < arraysize; ++j)
            {
                if (i == j)
                  numer[i][i] = cassign (1.+old[i][i].re, old[i][i].re);
                else
                  numer[i][j] = cassign (old[i][j].re, old[i][j].im);
            }

        cmatvect_mult (numer, steer, weights, arraysize);
        /* ---------------- COMPLETE WEIGHT COMPUTATIONS ---------------- */
        for (i = 0; i < arraysize; ++i)
        {
            weights[i].re /= denom;
            weights[i].im /= denom;
        }
}

/** FOLLOWING ROUTINES  FOR OUTPUTTING GRAPH TO HP SCREEN **/

Beamform (weights, arraysize, pattern)
 int                arraysize;
 complex_vector     weights;
 float              pattern[];
{
        double      cos (),  sin (),  sqrt ();
        int         i, angle;
        float       theta, number, angle2;
        complex     sum;

        /* ---------------------- ANGLE SPANS  -pi/2 TO pi/2 ---------------------- */
        for (angle = -180; angle <= 180; ++angle)
        {
            sum = cassign (0.,0.);
            for (i = 0; i < arraysize; ++i)
            {
                theta = 2. * M_PI * i * spacing * sin ((angle/2.) degrad);
                theta /= wavelength;
                /*  FOR ALL HYDROPHONES: WEIGHTS ADDED TOGETHER  */
                sum.re += weights[i].re*cos(theta) - weights[i].im*sin(theta);
                sum.im += weights[i].re*sin(theta) + weights[i].im*cos(theta);
            }
            /* ---- MAGNITUDE OF SUM (OF WEIGHTS) IS CALCULATED --- */
            pattern[angle+180] = sqrt ((sum.re * sum.re) + (sum.im * sum.im));
        }
}
```

```
Normalize (weights, array_size, pattern, normalized_pattern)
 int                array_size;
 complex_vector     weights;
 float              pattern[];
 float              normalized_pattern[];
{
      double              sqrt (),  log10 ();
      int                 i, angle;
      float               constant = 0., mag;

      for (i = 0; i < array_size; ++i)
      {
            mag = cabs (weights[i]);
            constant += mag;
      }
      for (angle = 0; angle <= 360; ++angle)
            normalized_pattern[angle] = 20. * log10 (pattern[angle]/constant);
}



Output (file, normalized_pattern)
 char       *file;
 float      normalized_pattern[];
{
      int               angle;
      static int        firsttime = 0;
      float             angle2;
      FILE              *datafile, *fopen ();

      datafile  =  fopen (file, "w");
      for (angle = -180; angle <= 180; ++angle)
      {
            if (normalized_pattern[angle+180] < -50.)          /* CHECK IF OFF GRAPH */
            normalized_pattern[angle+180] = -50.;
            /*  OUTPUT BOTH ANGLE & DECIBEL VALUE AT THAT ANGLE  */
            angle2 = angle / 2.;
            fprintf (datafile, "%f %f\n", angle2, normalized_pattern[angle+180]);
      }
      fclose (datafile);
}

/*  ************************** MAIN ROUTINE ************************** */

main ()
{
      FILE              *fopen (), *data;
      int               i, j, arraysize, sources, dominant, iam;
      float             pattern[400], npattern[400];
      char              *filename;
      vector            sigvals, bears;
      complex_matrix    eigvects, eigvalues;
      complex_vector    inputvector, steering, weights;
```

```
    kinit(100);                          /* ATTACH TO THE TOPOLOGIX KERNEL */

    /* ---------- GET VIRTUAL NODE ID FOR COMPUTATIONS ---------- */
    iam = v16_node_number[getnodeid()];

    /* --------- DEFINE NUMBER OF HYDROPHONES IN ARRAY --------- */
    arraysize = 32;
    if (iam == 0)
    {
        printf ("\n\tHow many sources are present?\n");
        scanf ("%d", &sources);
        data = fopen ("DATAXX", "r");
        for (i = 0; i < arraysize; ++i)
            for (j = 0; j < arraysize; ++j)
                fscanf (data,"%f %f",&eigvalues[i][j].re,&eigvalues[i][j].im);
        fclose (data);
    }

    /* ---- INITIALIZE EIGENVECTORS TO I AND WEIGHTS TO ONE ----- */
    for (i = 0; i < arraysize; ++i)
    {
        weights[i] = cassign (1., 0.);
        for (j = 0; j < arraysize; ++j)
        {
            if (i == j)
                eigvects[i][i] = cassign (1., 0.);
            else
                eigvects[i][j] = cassign (0., 0.);
        }
    }

    /* -------- READ IN VALUES NEEDED FOR PROGRAM TO RUN -------- */
    data = fopen ("values", "r");
    Inputvalues (data, sigvals, bears, sources);
    fclose (data);
    for (i  =  0; i < 10; ++i)
    {
        /* ---------------- GENERATE NEW DATA VECTOR ---------------- */
        ProcessInputs (arraysize, sources, sigvals, bears, i, inputvector);
        /* ------- UPDATE THE EIGENVECTORS WITH NEW DATA -------- */
        Update_data (eigvalues, eigvects, inputvector, .8188, arraysize);
        /* - PERFORM QR DECOMPOSITION AND RQ MULTIPLICATION -
*/
        QR_Iteration (eigvalues, eigvects, arraysize);
    }
    /* ---------------- GENERATE THE STEERING VECTOR ---------------- */
    SteerGen (arraysize, 0., steering);
    /* -- PERFORMEMVDR ALGORITHM WITH CALCULATED VALUES --- */
    dominant = 5;
    EMV (dominant, 1., arraysize, eigvects, eigvalues, steering, weights);

    kexit(10);                           /* REMOVE FROM TOPOLOGIX KERNEL */
} /* ---------------------- END OF THE MAIN PROGRAM ---------------------- */
```

APPENDIX B

SOURCE CODE FOR BROADCAST AND EXCHANGE ROUTINES

```
/*  ************************************************************************
    THE TWO ROUTINES IN THIS FILE ARE THE COMMUNICATION
    ROUTINES THAT ARE NEEDED BY PROGRAMS THAT PASS DATA
    ON THE TOPOLOGIX SYSTEM.
************************************************************************  */
static int Which_Port[4] = { DL_CB0, DL_CB1, DL_CB2, DL_CB3};

Broadcast (data, size, whohasthedata)
 int  *data, size, whohasthedata;
/*  ------------------------------------------------------------------------
    THIS ROUTINE WILL BROADCAST DATA THROUGH THE
    HYPERCUBE.  THE DATA IS ORIGINALLY HELD BY ONE PROCESSOR.
    THIS ROUTINE CAN PERFORM THE BROADCAST IN  O(d)
    OPERATIONS.

    INPUT:
            data :     type pointer
                       Points to data that will be broadcast.

            size :     type integer
                       Size of the data block.

    whohasthedata :    type integer
                       Hypercube processor id
------------------------------------------------------------------------  */
{
    int  bit, whoiam=getnodeid(), mask=0xFFFF, orbit=0x1, whofrom;
    int  iamwho = whoiam;

    /* ---------- SET MESSAGE FLAGS NEEDED ---------- */
    mesg.nh_type = 0;
    mesg.nh_length = size;
    mesg.nh_flags = 0;
    mesg.nh_node = 0;
    mesg.nh_msg = data;

    for (bit = 0; bit < 4; ++bit) /* COVER ALL BITS IN ID */
    {
        whoiam &= mask;/* MASK OFF UNNEEDED BITS FROM SOURCE ID */
        whohasthedata &= mask;/* MASK OFF UNNEEDED BITS */
        mask <<= 1;/* SHIFT MASK BIT OVER BY ONE TO LEFT */
        whofrom = whoiam ^ orbit;
        orbit <<= 1;/* SHIFT ORBIT BIT OVER BY ONE TO LEFT */

        mesg.nh_dl_event = Which_Port[bit];
        mesg.nh_event = abs (Which_Port[bit]);

        if (whoiam == whohasthedata)
            dsend (&mesg);
        else if (whofrom == whohasthedata)
            drecv (&mesg);
    } /* ---------- END OF THE FOR LOOP ---------- */
} /* ---------- END OF THE PROGRAM ---------- */
```

```
Exchange (Xmatrix, size, matsize)
 int                 size, matsize;
 complex_matrix      Xmatrix;
/*  ------------------------------------------------------------------------------
     THIS ROUTINE EXCHANGES DATA HELD IN ALL PROCESSORS
     WITH ALL THE PROCESSORS.

     INPUT:
          Xmatrix :    type complex_matrix
                       All vectors but the ones to pass should be set to zero.

              size :   type integer
                       Specifies size of Xmatrix in bytes.

           matsize :   type integer
                       Specifies # of rows in Xmatrix.

     OUTPUT:
          Xmatrix :    type complex_matrix
                       Contains all data from all procs.
------------------------------------------------------------------------------  */
{
     int                 bit, iam = getnodeid(), orbit = 1, i, j;
     complex_matrix      X2matrix;
     static struct nmsg  mesg2;

     mesg.nh_type = mesg2.nh_type = 0;
     mesg.nh_length = mesg2.nh_length = size;
     mesg.nh_flags = mesg2.nh_flags = 0;
     mesg.nh_node = mesg2.nh_node = 0;
     for (bit = 0; bit < 4; ++bit)
     {
          mesg.nh_event = abs (Which_Port[bit]);
          mesg2.nh_event = abs (Which_Port[bit]);
          if ((iam & orbit) != 0) /* PLANE ONE */
          {
               mesg.nh_msg = Xmatrix;
               mesg.nh_dl_event = Which_Port[bit];
               dsend (&mesg);
               mesg2.nh_msg = X2matrix;
               drecv (&mesg2);
          }
          else  /* ------------------------- PLANE TWO ------------------------- */
          {
               mesg2.nh_msg = X2matrix;
               drecv (&mesg2);
               mesg.nh_msg = Xmatrix;
               mesg.nh_dl_event = Which_Port[bit];
               dsend (&mesg);
          }
          cmat_add (Xmatrix, X2matrix, matsize); /* PERFORM DATA EXCHANGE */
          orbit <<= 1;
     }
}
```

INITIAL DISTRIBUTION LIST

Addressee		No. of Copies
NAVSEA	(PMS-412 (Capt. Tuma, LCDR Kasputis, B. Zarnich, E. Neuman, P. Mansfield, P. Imbert, Y. Dogrul))	7
NOSC	(Code 741 (G. Byram, S. I. Chou))	2
NRL	(Code 5155 (R. Hillson))	1
NADC	(Code 5021 (L. Hart), Code 5052 (J. Whalon), Code 503 (P. Santi))	3
NWSC	(Code 6044 (G. Summerville))	1
DARPA, NTO	(Dr. C. Stuart, Dr. G. Mohnkern)	2
ONR	(Code 1114 (R. N. Madan))	1
DTIC		12